

MEMOIZATION AND DYNAMIC PROGRAMMING

Reading: CLR Chapter 16

Below is a recursive function `Raise2` that computes 2^n .

```
Raise2(n)
  if n = 0 then
    return 1
  else
    return 2 * Raise2(n-1)
```

For any input, can draw an *invocation tree* in which each node is labeled by `Raise2(i)` for some i , and each `Raise2(i-1)` is a child of `Raise2(i)`. E.g. Draw the invocation tree for `Raise2(3)`:

Suppose we had a machine that didn't have a multiply operator. Then we might have

```
Raise2-Slow(n)
  if n = 0 then
    return 1
  else
    return Raise2-Slow(n-1) + Raise2-Slow(n-1)
```

What is the recurrence relation and solution for the running-time of `Raise2-Slow`?

Draw the invocation tree for `Raise2-Slow(3)`:

The reason `Raise2-Slow` is so slow is that it re-solves the same subproblems many times. We can make it fast again by remembering the result of a subproblem once it is solved. This effectively "glues" together nodes with the same label in the function call tree to form a DAG (Directed Acyclic Graph).

In general, we can use an auxiliary table to remember previously computed results and get the effect of computing with a DAG rather than a tree. Here's how we do this for `Raise2`:

```
Raise2-Fast(n)
  T <- new array[0..n]
  for i <- 0 to n do
    T[i] <- 0 {0 is an "illegal" result indicating an empty slot}
  return Raise2-Memo(T,n)

Raise2-Memo(T,n)
  if T[n] = 0 then
    {Calculate result first time and remember it in T}
    if n = 0 then
      T[n] <- 1
    else
      T[n] <- Raise2-Memo(T,n-1) + Raise2-Memo(T,n-1)
    {Look up previously computed result in T}
  return T[n]
```

This strategy of remembering the results of subproblems in a table is called **memoization** (not memorization!)

In the case of `Raise2`, a table is not actually necessary; we can remember the subproblem result in a local variable instead:

```
Raise2-Fast1(n)
  if n = 0 then
    return 1
  else
    subresult <- Raise2-Fast1(n-1)
  return subresult + subresult
```

Fibonacci Numbers

Using memoization for `Raise2` seems like overkill, especially when there is a simpler way to achieve the same result without a memoization table (i.e., using a local variable). An example where memoization is a clearer "win" is calculating Fibonacci numbers. Here is the naive recursive function for calculating these:

```
Fib-Slow(n)
  if n <= 1 then
    n
  else
    Fib-Slow(n-1) + Fib-Slow(n-2)
```

Draw an invocation tree for `Fib(4)`:

Here's the result of applying the memoization strategy to Fib-Slow :

```
Fib-Fast(n)
  T <- new array[0..n]
  for i <- 0 to n do
    T[i] <- -1 {-1 is an "illegal" result indicating an empty slot}
  return Fib-Memo(T,n)

Fib-Memo(T,n)
  if T[n] < 0 then
    {Calculate result first time and remember it in T}
    if n <= 1 then
      T[n] <- n
    else
      T[n] <- Fib-Memo(T,n-1) + Fib-Memo(T,n-2)
    {Look up previously computed result in T}
  return T[n]
```

Draw how the calculation proceeds for Fib-Fast(4):

The key aspect of the memoized Fibonacci function is that it uses the table T to avoid recomputation. Rather than keeping the recursive structure of the naive Fibonacci, we can instead organize the process around filling in the slots of T. The strategy of organizing a computation around a table that avoids recomputation is called **dynamic programming**. Here is a dynamic programming solution to Fibonacci:

```
Fib-DP(n)
  T <- new array[0..n]
  {By dependencies of Fibonacci, fill in slots from low to high.}
  T[0] = 0
  T[1] = 1
  for i <- 2 to n do
    T[i] <- T[i-1] + T[i-2]
  return T[n]
```

In the case of Fibonacci, it's not necessary to use a table whose size is linear in n -- we can get by with a constant number of variable slots. How many?

Pascal's Triangle

Recall Pascal's triangle. Each element is the sum of the two elements above it, except for edge elements, which are 1:

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1

```

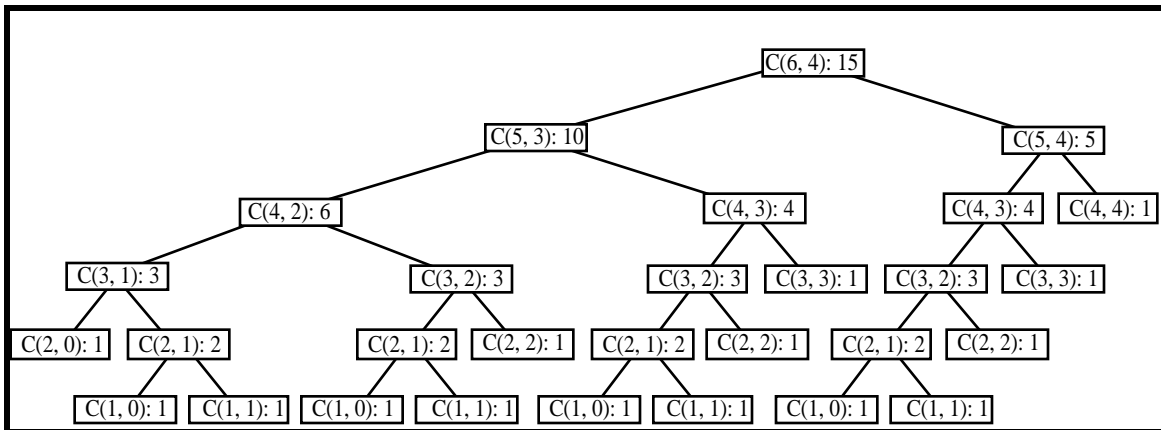
The following function computes the k th element of the n th row (k and n are 0-based):

```

Pascal(n,k)
  if k = 0 or k = n then
    return 1
  else
    return Pascal(n-1,k-1) + Pascal(n-1, k)

```

Below is an invocation tree for $\text{Pascal}(6, 4)$ in which each call $\text{Pascal}(n, k)$ has been abbreviated $C(n, k):r$, where r is the result returned by the call on arguments (n, k) .



In the worst case, $C(n, k)$ can take time exponential in n . Note that the exponential nature is due to subproblem duplication, which can be removed by memoization. We use a two-dimensional table $P[i, j]$ to store previously computed results:

```

Pascal-Fast(n,k)
  P <- new array[0..n][0..k] {Can make table smaller; see below}
  for i <- 0 to n do
    for j <- 0 to k do
      P[i,j] <- 0 {0 is an "illegal" result indicating an empty slot}
  Pascal-Memo(P,n,k)

```

```

Pascal-Memo(P,n,k)
  if P(n,k) = 0 then
    if k = 0 or k = n then
      P[n,k] <- 1
    else
      P[n,k] <- Pascal-Memo(P,n-1,k-1) + Pascal-Memo(P,n-1,k)
  return P(n,k)

```

An element is stored into each array element at most twice: once during initialization and at most once during Pascal-Memo. The running time of Fast-Pascal is therefore (n^2) .

We can be cleverer by initializing the top and left edge to 1 to avoid the inner `if` within Pascal-Memo. We can also be cleverer by not creating so large a table ($[0..n] \times [0..k]$) -- we only need a $[0..(n-k)] \times [0..k]$ table, although we have to change what the first index means to achieve this.

We can also construct a dynamic programming version that avoids the recursive control structure altogether. We must be careful to fill the slots in an order consistent with the dependencies implied by Pascal:

```
Pascal-Fast(n,k)
  P <- new array[0..(n-k)][0..k]
  for i <- 0 to (n-k) do
    P[i,0] = 1 {Base case}
  for j <- 1 to k do
    P[0, j] = 1 {Base case}
    for i <- 1 to (n-k) do
      P[i,j] = P[i-1,j-1] + P[i-1,j]
  return P[n-k, k]
```

Longest Common Subsequence

Want to find the longest common subsequence (LCS) that is shared among two sequences. E.g. the longest common subsequence of (B, A, C, B) and (C, A, B, A, B) is (B, A, B). Often the LCS is not unique. E.g. for (B, A, C, B) and (A, B, C, A, B), both (A, C, B) and (B, A, B) are LCSs.

There is a simple but exponential-time algorithm for computing the LCS of two linked lists:

```
LCS(s,t) =
  if empty?(s) or empty?(t) then
    return empty()
  else if head(s) = head(t) then
    prepend(head(s), LCS(tail(s), tail(t)))
  else
    L1 <- LCS(s, tail(t))
    L2 <- LCS(tail(s), t)
    if length(L1) >= length(L2) then
      return L1
    else
      return L2
```

By using a two-dimensional table indexed by the lists, can avoid the exponential recomputation and get a $\Theta(mn)$ running time, where m and n are the lengths of the two lists.

Example:

	(A,B,C,A,B)	(B,C,A,B)	(C,A,B)	(A,B)	(B)	()
(B, A, C, B)						
(A, C, B)						
(C, B)						
(B)						
()						

Can fill the table in either recursively (memoization) or iteratively (dynamic programming). Recursion has the overhead of recursion, but computes fewer entries in the table. Dynamic programming avoids recursive overhead, but computes more entries than it has to.